

# Seguridad en APIs RESTful y mitigación de vulnerabilidades OWASP

## Objetivo del Ejercicio:

El objetivo de este ejercicio fue analizar una API vulnerable en PHP, detectar fallos relacionados con el OWASP Top 10 (específicamente Broken Access Control e Inyección SQL), y aplicar buenas prácticas para mitigar estos riesgos mediante la reescritura de código seguro.

## Escenario Analizado:

Se proporcionó un fragmento de una API en PHP que expone un endpoint para eliminar usuarios, con el siguiente código vulnerable:

```
<?php
$id = $_GET['id'];
if ($_GET['role'] === 'admin') {
    $conexion->query("DELETE FROM usuarios WHERE id=$id");
    echo "Usuario eliminado";
} else {
    echo "Acceso denegado";
}
?>
```

## 1. Análisis de Vulnerabilidades

Se identificaron las siguientes vulnerabilidades críticas en el código proporcionado:

- **Inyección SQL:** La variable `$id` se concatena directamente en la consulta SQL, permitiendo a un atacante insertar código SQL malicioso y alterar la consulta, pudiendo borrar datos arbitrarios o toda la tabla.
- **Broken Access Control (Control de Acceso Roto):** La verificación del rol se basa en un parámetro `$_GET['role']` directamente desde la URL. Esto es altamente inseguro, ya que un atacante puede manipular fácilmente este parámetro para simular ser un administrador sin una autenticación o autorización real.
- **Falta de autenticación/autorización robusta:** No existe un mecanismo de autenticación (como una sesión o JWT) que verifique la identidad del usuario. La autorización del rol se basa en una entrada de usuario fácilmente manipulable.

**Impacto para un atacante si conoce este endpoint:** Si un atacante conociera este endpoint, podría:

- **Eliminar usuarios sin estar autenticado:** Manipulando el parámetro `$_GET['role']` a 'admin', el atacante podría ejecutar la función de eliminación.
- **Manipular el id en la URL para borrar usuarios arbitrarios:** Utilizando la vulnerabilidad de Inyección SQL, un atacante podría modificar el parámetro `$_GET['id']` para borrar usuarios específicos, múltiples usuarios, o incluso toda la base de datos (ej., `id=1 OR 1=1`).

## 2. Reescritura Segura del Código

Se reescribió el código vulnerable aplicando las siguientes medidas de seguridad:

1. **Verificación por JWT o Sesión Autenticada:** Se reemplazó la dependencia de `$_GET['role']` por una simulación de un rol obtenido de una fuente segura (`$user_role_secure`). En una aplicación real, esto se obtendría de una sesión validada o de un token JWT decodificado y verificado en el servidor.
2. **Consultas Preparadas:** Se implementaron consultas preparadas (`mysqli_prepare()` y `mysqli_bind_param()`) para la operación DELETE. Esto asegura que los valores de entrada del usuario (`$id_to_delete`) sean tratados como datos y no como parte de la lógica SQL, previniendo la Inyección SQL.
3. **Verificación de Rol Robusta:** Se verifica que el usuario autenticado (simulado a través de `$user_role_secure`) tenga el rol de 'admin' para ejecutar la eliminación, garantizando que la autorización se base en una fuente segura.

```
$user_role_secure = 'admin'; // <--- CAMBIA ESTO PARA PROBAR DIFERENTES ROLES SEGUROS
// Obtener el ID del usuario a eliminar de forma segura
$id_to_delete = $_GET['id'] ?? null;
// Validar que el ID es un número entero (capa adicional de seguridad)
if (!filter_var($id_to_delete, FILTER_VALIDATE_INT) || $id_to_delete !== null) {
    echo "ID de usuario inválido.";
    $conexion->close();
    exit();
}
// 1. y 3. Verificar que el usuario tenga el rol correcto desde una fuente segura
// y que el ID no sea nulo.
if ($user_role_secure === 'admin' && $id_to_delete !== null) {
    // 2. Usar consultas preparadas para eliminar el usuario y prevenir Inyección SQL
    $stmt = $conexion->prepare("DELETE FROM usuarios WHERE id = ?");
    if ($stmt === false) {
        die("Error al preparar la consulta: " . $conexion->error);
    }
    // "i" indica que el parámetro es de tipo entero (para un ID numérico)
    $stmt->bind_param("i", $id_to_delete);
    if ($stmt->execute()) {
        if ($stmt->affected_rows > 0) {
            echo "Usuario con ID " . htmlspecialchars($id_to_delete) . " eliminado exitosamente.";
        } else {
            echo "No se encontró ningún usuario con ID " . htmlspecialchars($id_to_delete) . " o ya fue eliminado.";
        }
    } else {
        echo "Error al eliminar el usuario: " . $stmt->error;
    }
    $stmt->close(); // Cerrar el statement
} else {
    // Si el rol no es 'admin' o no se proporcionó un ID válido
    echo "Acceso denegado o ID de usuario no proporcionado/inválido.";
}
$conexion->close(); // Cerrar la conexión a la base de datos
?>
```

## Código Seguro Implementado:

```
<?php
include 'conexion.php'; // Archivo de conexión a la base de datos

// SIMULACIÓN DE ROL SEGURO: En producción, esto vendría de una sesión o JWT
$user_role_secure = 'admin'; // Cambiar para probar diferentes roles (ej. 'user')

$id_to_delete = $_GET['id'] ?? null;

// Validar que el ID es un número entero
if (!filter_var($id_to_delete, FILTER_VALIDATE_INT) && $id_to_delete !== null) {
    echo "ID de usuario inválido.";
    $conexion->close();
    exit();
}

if ($user_role_secure === 'admin' && $id_to_delete !== null) {
    $stmt = $conexion->prepare("DELETE FROM usuarios WHERE id = ?");

    if ($stmt === false) {
        die("Error al preparar la consulta: " . $conexion->error);
    }

    $stmt->bind_param("i", $id_to_delete); // "i" para entero

    if ($stmt->execute()) {
        if ($stmt->affected_rows > 0) {
            echo "Usuario con ID " . htmlspecialchars($id_to_delete) . " eliminado
exitosamente.";
        } else {
            echo "No se encontró ningún usuario con ID " . htmlspecialchars($id_to_delete) . " o
ya fue eliminado.";
        }
    } else {
        echo "Error al eliminar el usuario: " . $stmt->error;
    }
    $stmt->close();
} else {
    echo "Acceso denegado o ID de usuario no proporcionado/inválido.";
}

$conexion->close();
?>
```

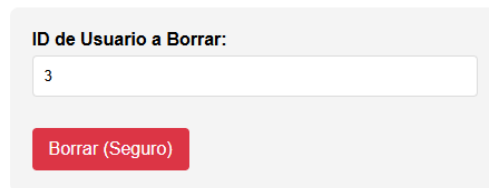
### 3. Aplicación Práctica

Se realizó la simulación de los endpoints (vulnerable y seguro) utilizando un index.html básico para enviar solicitudes GET y verificar los resultados en el navegador.

#### Resultados de la Simulación:

- **Versión Vulnerable:**

- Al enviar id=X&role=admin, el usuario era efectivamente eliminado, demostrando el **Broken Access Control**.

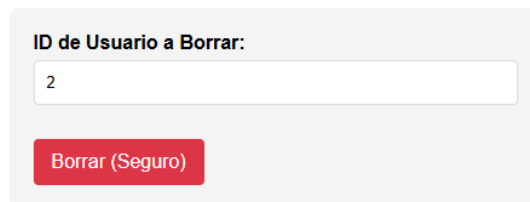


The screenshot shows a web form with the label "ID de Usuario a Borrar:". Below the label is a text input field containing the number "3". Below the input field is a red button with the text "Borrar (Seguro)".

Usuario eliminado

- **Versión Segura:**

- **Con rol 'admin' (simulado):** La eliminación del usuario especificado funcionó correctamente, confirmando que la lógica de borrado y las consultas preparadas operaban como se esperaba.



The screenshot shows a web form with the label "ID de Usuario a Borrar:". Below the label is a text input field containing the number "2". Below the input field is a red button with the text "Borrar (Seguro)".

Usuario con ID 2 eliminado exitosamente.

- **Con rol 'user' (simulado):** Al cambiar la variable `$user_role_secure` a 'user' y intentar borrar, el sistema respondió con "Acceso denegado o ID de usuario no proporcionado/inválido", demostrando el control de acceso basado en rol seguro.

```
// diferentes roles (ej. 'admin', 'user',  
$user_role_secure = 'user'; // <--- CAMBIA
```

ID de Usuario a Borrar:

Borrar (Seguro)

Acceso denegado o ID de usuario no proporcionado/inválido.

## Reflexión:

- **Ventaja de usar roles desde sesión en vez de parámetros GET:**
  - La principal ventaja es la **seguridad y la confiabilidad**. Los parámetros GET son fácilmente manipulables en la URL, lo que permite a un atacante cambiar su rol y obtener privilegios no autorizados. Un rol gestionado desde una sesión segura (o un JWT validado y firmado en el servidor) garantiza que la autorización se base en una fuente de verdad verificada por el backend, no en una entrada del cliente.
  - Esto previene el **Broken Access Control**, una vulnerabilidad crítica del OWASP Top 10.
- **Peligro de construir consultas directamente con valores de entrada del usuario:**
  - Es extremadamente peligroso porque expone la aplicación a **Inyección SQL**. Al concatenar directamente los valores de entrada, un atacante puede insertar comandos SQL maliciosos que el servidor interpreta y ejecuta como parte de la consulta original.
  - Esto puede llevar a lectura de datos sensibles, modificación o eliminación de información, e incluso la ejecución remota de código, comprometiendo gravemente la integridad y confidencialidad de la base de datos.
  - Las **consultas preparadas** son la defensa estándar contra este tipo de ataque, ya que separan los datos de la lógica de la consulta SQL.